Data Sci. Eng. (2016) 1:219–230
DOI 10.1007/s41019-017-0033-5

Efficient Maximal Clique Enumeration Over Graph Data

Boyi Hou[1] · Zhuo Wang[1] · Qun Chen[1] · Bo Suo[1] · Chao Fang[1] · Zhanhuai Li[1] · Zachary G. Ives[2]

Received: 1 July 2016 / Accepted: 14 December 2016 / Published online: 7 February 2017

Abstract In a wide variety of emerging data-intensive applications, such as social network analysis, Web document clustering, entity resolution, and detection of consistently co-expressed genes in systems biology, the detection of *dense subgraphs* (cliques) is an essential component. Unfortunately, this problem is NP-Complete and thus computationally intensive at scale—hence there is a need for efficient processing, as well as the techniques for distributing the computation across multiple machines such that the computation, which is too time-consuming on a single machine, can be efficiently performed on a machine cluster given that it is large enough. In this paper, we propose a new algorithm (called GP) for maximal clique enumeration. It identifies cliques by the operation of binary graph partitioning, which iteratively divides a graph until each task is sufficiently small to be processed in parallel. Given a connected graph $G = (V, E)$, the GP algorithm has a space complexity of $O(|E|)$ and a time complexity of $O(|E|\mu(G))$, where $\mu(G)$ represents the number of different cliques existing in G. We also present a hybrid algorithm, which can effectively leverage the advantages of both the GP algorithm and the classical Bron-and-Kerbosch (BK) algorithm. Then, we develop corresponding parallel solutions based on the GP and hybrid algorithms. Finally, we evaluate the performance of the proposed solutions on real and synthetic graph data. Our extensive experiments show that in both centralized and parallel setting, our proposed GP and hybrid approaches achieve considerably better performance than the state-of-the-art BK approach. Our parallel solutions are implemented and evaluated on MapReduce, a popular shared-nothing parallel framework, but can easily generalize to other shared-nothing or shared-memory parallel frameworks.

Keywords Maximal clique enumeration · Parallel graph processing · Iterative graph partitioning · MapReduce

✉ Qun Chen
chenbenben@nwpu.edu.cn

Boyi Hou
byhou.mail@nwpu.edu.cn

Zhuo Wang
wzhuo918@mail.nwpu.edu.cn

Bo Suo
caitou@mail.nwpu.edu.cn

Chao Fang
cfang.mail@nwpu.edu.cn

Zhanhuai Li
lizhh@nwpu.edu.cn

Zachary G. Ives
zives@cis.upenn.edu

[1] School of Computer Science, Northwestern Polytechnical University, Xi'an, China

[2] Department of Computer and Information Systems, University of Pennsylvania, Philadelphia, PA, USA

1 Introduction

A variety of emerging applications are focused on computations over data modeled as a graph: examples include finding groups of actors or communities in social networks [18, 22], Web mining [19], entity resolution [26], graph mining [37, 41], and detection of consistently co-expressed gene groups in systems biology [27]. For the problems just cited, as well as a number of others, a critical component of the analysis is the detection of cliques (fully connected components) in the structure of the network graph.

Maximal clique enumeration is NP-Complete. Hence, a great deal of effort has been spent on efficient search

algorithms [1, 4, 6, 14, 33, 34]. Most of existing algorithms for maximal clique enumeration are based on the classical BK algorithm proposed by Bron and Kerbosch [4], which uses a backtracking technique to explore search space and limits the size of its search space by remembering the search paths it has already visited. A variant [34] of the BK algorithm also provides a worst-case optimal solution. In practice, the BK algorithm has been widely reported as being faster than its alternatives [5, 15].

Data-intensive applications usually require clique detection to be operated over large graphs. We observe that the existing algorithms were optimized for centralized implementation, but not for parallel implementation. Their performance has not been adequately evaluated on real big graphs either, especially the natural graphs with the skewed power-law degree distributions commonly found in real world. In fact, as we show in experimental evaluation of Sect. 5, their performance is quite sensitive to particular graph characteristics. We also note that there have been a variety of proposals that divide the graph into smaller subcomponents and exploit parallelism to improve performance [10, 23, 32, 38, 40]. They have been empirically shown to speed computation in massive networks. However, built on the BK algorithm, their performance may be limited by the efficiency of *BK* search and how evenly a graph is partitioned.

In this paper, we present a new approach for maximal clique enumeration. Versus prior work in this area, its key insight is to exploit iterative binary decomposition during the computation. It iteratively divides a graph until each task is sufficiently small to be processed in parallel. As a result, a computation, which may be too time-consuming on a single machine, can be effectively parallelized across a cluster. In this paper, we choose MapReduce for parallel evaluation due to the maturity and wide availability of its implementations. However, the implementation can easily generalize to other shared-nothing or shared-memory parallel architectures. The major contributions of this paper are summarized as follows:

1. We present a novel algorithm (GP) for maximal clique enumeration based on iterative binary graph partitioning. Given a connected graph $G = (V, E)$, it has the space complexity of $O(|E|)$ and the time complexity of $O(|E|\mu(G))$, where $\mu(G)$ represents the number of different cliques existing in G.
2. We propose a hybrid algorithm for maximal clique enumeration, which can effectively leverage the advantages of both GP and BK algorithms.
3. We develop parallel solutions to maximal clique enumeration based on the GP and hybrid algorithms and implement them on MapReduce. By using binary graph partitioning to divide the tasks, the proposed solutions can effectively parallelize maximal clique computation with improved load balancing.
4. We experimentally evaluate the performance of our proposed solutions over a wide variety of graph data available in open source. Our extensive experiments show that in both centralized and parallel settings, our proposed GP approach achieves considerably better performance than the state-of-the-art BK approach and the hybrid approach performs better than both of them.

Note that this paper is an extension of our preliminary work published in [7]. The major new contribution of this extended work is the hybrid approach that can achieve better performance than both BK and GP. The rest of this paper is organized as follows: Sect. 2 provides the background information and briefly describes the existing techniques. Section 3 presents the GP and hybrid algorithms. Section 4 presents our parallel solutions and their MapReduce implementation. Section 5 empirically evaluates the performance of the proposed solutions. Section 6 discusses related work. Finally, Sect. 7 concludes this paper.

2 Preliminaries

2.1 Definition: Clique and Maximal Clique

A clique is a subgraph in which every pair of vertices is connected by an edge. The definition of a *maximal* clique is as follows:

Definition 1 A maximal clique in a graph G is a clique not contained by any other clique in G.

The problem of maximal clique enumeration refers to identifying all the maximal cliques in a given graph G. Since each connected component in G can be processed independently, we assume that G is a connected graph in this paper.

2.2 Background: MapReduce

The MapReduce model processes distributed data across many nodes via three basic phases. In the Map phase, it takes an input and produces a list of intermediate key/value pairs without communication between nodes. Next, the Shuffle phase repartitions these intermediate pairs according to their keys across nodes. Finally, the Reduce phase aggregates the intermediate pairs it receives to produce final results. This process can be repeated by invoking an arbitrary number of additional Map-Shuffle–Reduce cycles as necessary.

In this paper, we use Hadoop for parallel evaluation and develop corresponding MapReduce solutions, in which graph partitioning is programmed in the Reduce phase.

2.3 Classical Sequential Algorithms

Algorithm 1: enumerateBK(anchor,cand,not)

```
1  if (cand = ∅) then
2      if (not = ∅) then
3          Output anchor;
4  else
5      fix_v ← the vertex in cand that is connected to the
       greatest number of other vertices in cand;
6      cur_v ← fix_v;
7      while (cur_v ≠ NULL) do
8          n_not← all the vertices in not that are connected to
           cur_v;
9          n_cand← all the vertices in cand that are connected
           to cur_v;
10         n_anchor←sub + {cur_v};
11         enumerateBK(n_anchor,n_cand,n_not);
12         not←not+ {cur_v};
13         cand←cand - {cur_v};
14         if (there is a vertex v in cand that is not connected to
           fix_v) then
15             cur_v ← v;
16         else
17             cur_v ← NULL;
```

For maximal clique enumeration, the BK algorithm [4] has been widely reported as being faster in practice than its alternatives [15, 32]. It is in essence a depth-first search, augmented with pruning tricks. Given a current vertex v and a set of candidate vertices S, it iteratively chooses a vertex u in S such that $N(u)$ has the biggest intersection set with S, in which $N(u)$ represents the set of u's neighboring vertices in S. When the candidate set S becomes empty, the algorithm outputs corresponding cliques and backtracks. It recursively traverses a search tree, performing the operations of vertex selection, set update, clique generation and backtracking.

The BK algorithm can be sketched by Algorithm 1. It uses three vertex sets to represent a search subtree: the set `anchor` records the list of vertices in the current search path, the set `cand` records the list of candidate vertices that are not in `anchor` but connected to every vertex in `anchor`, and the set `not` records the list of vertices that are connected to every vertex in anchor but could not produce new maximal cliques if combined with the vertices in the anchor set.

2.4 Existing Parallel Solutions

In this subsection, we describe the idea behind the typical parallel approach [23, 32, 38] for maximal clique enumeration based on MapReduce. It enumerates maximal cliques for different vertices in a graph in parallel.

Given a graph G and a vertex v in G, the maximal cliques of the vertex v refer to the maximal cliques containing v in G. Note that a vertex v's maximal cliques are the induced subgraphs consisting of v and its neighboring vertices in G. The parallel search consists of two steps. In the first one, the parallel approach retrieves each vertex's neighboring information relevant to its clique computation. In the second step, it searches for each vertex's maximal cliques in parallel. For the computation on an individual vertex, it simply adopts the classical sequential algorithms (e.g., the BK algorithm).

In the typical approach, enumerating the maximal cliques of a vertex is supposed to be performed on a single machine. In case that the computation on an individual vertex is extremely time-consuming due to the large number of maximal cliques, it may become a parallel performance bottleneck. The method proposed in [32] can parallelize maximal clique enumeration on an individual vertex. It uses candidate path data structures to record the search progress such that any search subtree can be traversed independently. It achieves better load balancing by allowing a computing node to steal some tasks from others when becoming almost idle. The proposed load balancing technique was implemented by MPI, but can easily generalize to other shared-nothing parallel frameworks such as MapReduce. However, as we will show in Sect. 5, its parallel performance depends on the performance of the *BK* algorithm, and may be limited by size unevenness among search subtrees.

3 Sequential Algorithms

3.1 Idea: Graph Partitioning

We illustrate the idea behind the new sequential algorithms by an example. As shown in Fig. 1a, the graph G consists of the vertices, $\{v_1, v_2, v_3, v_4, v_5\}$. We randomly choose a vertex in G (e.g., v_1) as the partitioning anchor and partition G into two subgraphs G_1^+ and G_1^-. G_1^+ denotes the induced subgraph consisting of v_1 and its neighboring vertices in G, $\{v_1, v_2, v_3\}$. G_1^- denotes the induced subgraph of G consisting of all the vertices not in G_1^+, $\{v_4, v_5\}$, and their neighboring vertices in G, $\{v_2, v_3\}$. The subgraphs G_1^+ and G_1^- are shown in Fig. 1b, c, respectively. We observe that any maximal clique of G is an induced subgraph of either G_1^+ or G_1^-.

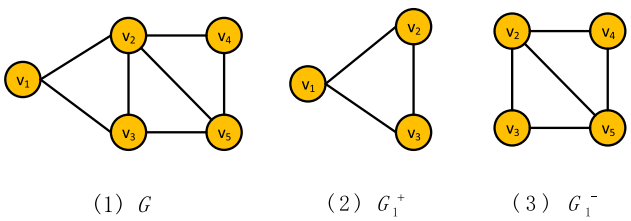


Fig. 1 A graph partitioning example

Generally, we have the following theorem:

Theorem 1 *Given a graph G, we partition G into two subgraphs, G_v^+ and G_v^-, in which v denotes a partitioning anchor, G_v^+ denotes the induced subgraph consisting of vertex v and its neighboring vertices in G, and G_v^- denotes the induced subgraph consisting of all the vertices not in G_v^+ and their neighboring vertices in G. Then, any maximal clique of G is an induced subgraph of either G_v^+ or G_v^-.*

Proof If a maximal clique contains the vertex v, it should be an induced subgraph of G_v^+. Otherwise, it should contain at least one vertex not in G_v^+. Suppose that it is the vertex u. As a result, the maximal clique is an induced subgraph of G_u, which consists of vertex u and its neighboring vertices. According to the definition of G_v^-, G_u is obviously an induced subgraph of G_v^-. Therefore, the maximal clique is an induced subgraph of G_v^-. □

According to Theorem 1, maximal clique detection in G can be performed by searching for the maximal cliques in G_v^+ and G_v^- independently. The partitioning operation can be iteratively invoked until all the resulting subgraphs become cliques. Obviously, all the maximal cliques in G are contained in the set of the resulting cliques. Unfortunately, a resulting clique generated by the above process cannot be guaranteed to be maximal. Therefore, enumeration algorithms should filter out the non-maximal cliques among them.

3.2 GP Algorithm

Algorithm 2: enumerateGP(anchor,cand,not)

```
1  if (G(cand) is a clique) then
2      Output the clique G(anchor ∪ cand);
3  else
4      while (G(cand) is NOT a clique) do
5          Choose a vertex v with the smallest degree in
           G(cand);
6          anchor+ ← anchor ∪ {v};
7          cand+ ← cand ∩ N(v);
8          not+ ← not ∩ N(v);
9          if (∄u ∈ not+ :u is connected to all the vertices in
           cand+) then
10             enumerateGP(anchor+,cand+,not+);
11         cand ← cand − {v};
12         not ← not ∪ {v};
13     if (∄u ∈ not:u is connected to all the vertices in cand)
       then
14         Output the clique G(anchor ∪ cand);
```

The algorithm iteratively partitions a graph until it becomes cliques. To reduce search space, it always chooses the vertex v with the smallest degree in a graph as the partitioning anchor. It can be observed that this strategy would usually result in a relatively small graph and a larger one. Generally, the small graph would be partitioned into cliques after only a few iterations, while the size of the larger one could be effectively reduced. Unlike the BK algorithm, which recursively extracts the induced subgraph consisting of the vertex with the largest degree and its neighbors, our approach instead recursively performs binary partitioning by choosing the partitioning anchor with the smallest degree.

The algorithm is sketched in Algorithm 2. Similar to the BK algorithm as shown in Algorithm 1, it employs three sets of vertices (anchor, cand and not) to record the partitioning progress and prune the subtrees that cannot generate maximal cliques. The recursive function first checks whether the resulting subgraph is a clique (Line 1). If yes, it simply outputs the subgraph. Otherwise, it chooses a partitioning anchor v with the smallest degree in cand and partitions $G(\texttt{cand})$ into $G(\texttt{cand}^+)$ and $G(\texttt{cand}^-)$. $G(\texttt{cand}^+)$ consists of v and its neighboring vertices in $G(\texttt{cand})$ (Lines 6–8). $G(\texttt{cand}^-)$ consists of all the vertices in $G(\texttt{cand})$ except v (Lines 11–12). The algorithm recursively processes the subgraph $G(\texttt{cand}^+)$ (Lines 9–10). Note that before the recursive function is invoked, the algorithm prunes the search space by inspecting whether there exists a vertex in the $\texttt{not}^+$ set that is connected to all the vertices in the $\texttt{cand}^+$ set (Line 9). Updating $G(\texttt{cand})$ with $G(\texttt{cand}^-)$ (Lines 11–12), it then iteratively invokes the partition operation to search for the maximal cliques in $G(\texttt{cand}^-)$ until $G(\texttt{cand}^-)$ becomes a clique (Lines 4–12). After $G(\texttt{cand}^-)$ becomes a clique, the algorithm checks whether it is maximal (Lines 13–14).

Given an input graph $G = (V, E)$, the algorithm can be set in motion by setting $\texttt{anchor} = \emptyset$, $\texttt{not} = \emptyset$ and $\texttt{cand} = V$. Suppose that we are running Algorithm 2 on the example graph as shown in Fig. 1. Originally, $\texttt{anchor} = \emptyset$, $\texttt{not} = \emptyset$ and $\texttt{cand} = \{v_1, v_2, v_3, v_4, v_5\}$. The vertex v_1 has the smallest degree of 2, is thus chosen as the partitioning anchor. G is then partitioned into G_1^+ and G_1^-. G_1^+ consists of v_1 and its neighboring vertices, $\{v_1, v_2, v_3\}$. G_1^- consists of the vertices, $\{v_2, v_3, v_4, v_5\}$. For G_1^+, $\texttt{anchor} = \{v_1\}$, $\texttt{not}=\emptyset$ and $\texttt{cand}=\{v_2, v_3\}$. For G_1^-, $\texttt{anchor}=\emptyset$, $\texttt{not}=\{v_1\}$ and $\texttt{cand}=\{v_2, v_3, v_4, v_5\}$. It can be observed that G_1^- is not a clique and v_3 has the smallest degree of 2 in G_1^-. G_1^- would then be partitioned into two subgraphs consisting of $\{v_2, v_3, v_5\}$ and $\{v_2, v_4, v_5\}$, respectively, which are both clique. Therefore, the maximal cliques of G can be computed with two partitioning operations.

In practical implementation, the algorithm iteratively partitions an input graph in a depth-first manner. After partitioning G into $G(\texttt{cand}^-)$ and $G(\texttt{cand}^+)$, it always processes the $G(\texttt{cand}^-)$ subgraph before $G(\texttt{cand}^+)$, and pushes the resulting $G(\texttt{cand}^+)$ into a stack for later processing. Whenever a $G(\texttt{cand}^-)$ subgraph becomes a

clique, it pops a $G(\texttt{cand}^+)$ subgraph from the stack and repeats the iterative partitioning operation.

We have Theorems 2 and 3, whose proofs are presented in "Appendices 1 and 2", respectively. Note that in Theorem 3, *different cliques* include both maximal and non-maximal cliques.

Theorem 2 *Algorithm 2 exactly returns all the maximal cliques in G.*

Theorem 3 *Given a connected graph $G = (V, E)$, Algorithm 2 has the space complexity of $O(|E|)$ and the time complexity of $O(|E|\mu(G))$, in which $\mu(G)$ represents the number of different cliques in G.*

3.3 Hybrid Algorithm

Algorithm 3: enumerateHybrid(anchor,cand,not)

```
1  if (G(cand) is a clique) then
2      Output the clique G(anchor ∪ cand);
3  else
4      while (G(cand) is NOT a clique) do
5          if (the largest vertex degree of G(cand) exceeds θ)
           then
6              v ← the vertex with the largest degree in
               G(cand);
7              cur_v=v;
8              while cur_v ≠ NULL do
9                  n_anchor=anchor ∪ {cur_v};
10                 n_not=not ∩ N(cur_v);
11                 n_cand=cand ∩ N(cur_v);
12                 if (∄u ∈ n_not:u is connected to all the
                   vertices in n_cand) then
13                     enumerateHybrid(n_anchor,n_cand,
                       n_not);
14                 not=not∪{cur_v};
15                 cand=cand − {cur_v};
16                 if (there is a vertex u in cand that is not
                   connected to v) then
17                     cur_v=u;
18                 else
19                     cur_v=NULL
20             return; // terminate this function;
21         else
22             Choose a vertex v with the smallest degree in
               G(cand);
23             anchor+ ← anchor ∪ {v};
24             cand+ ← cand ∩ N(v);
25             not+ ← not ∩ N(v);
26             if (∄u ∈ not+:u is connected to all the vertices
               in cand+) then
27                 enumerateHybrid(anchor+,cand+,not+)
                   cand ← cand − {v};
28                 not ← not ∪ {v};
29     if (∄u ∈ not:u is connected to all the vertices in cand)
       then
30         Output the clique G(anchor ∪ cand);
```

We observe that the performance of Algorithm 2 largely depends on the number of the generated $G(\texttt{cand}^+)$ subgraphs, on which the partition operation is iteratively executed. To reduce the number of invoked partition operations, the hybrid algorithm considers multi-way partitioning as well as binary partitioning. The operation of multi-way partitioning selects a vertex v with the largest degree in $G(\texttt{cand})$, and partitions $G(\texttt{cand})$ into $\{G_v(\texttt{cand}), G_{u_1}(\texttt{cand}), \ldots, G_{u_k}(\texttt{cand})\}$, in which $\{u_1, \ldots, u_k\}$ represent the set of vertices *not* connected to v in $G(\texttt{cand})$ and $G_v(\texttt{cand})$ denotes the induced subgraph consisting of the vertex v and all its neighbors in $G(\texttt{cand})$. It is worthy to point out that the multi-way partition operation is essentially the core search operation used by the classical BK algorithm. As proved in the BK algorithm, it can be shown that all the maximal cliques in $G(\texttt{cand})$ can be searched in the induced subgraphs of $\{G_v(\texttt{cand}), G_{u_1}(\texttt{cand}), \ldots, G_{u_k}(\texttt{cand})\}$ independently. Since this algorithm uses both binary and multi-way partitioning operations, which are the key characteristics of the GP and BK algorithms, respectively, it is called *Hybrid*.

The hybrid algorithm invokes the operation of multi-way partitioning if and only if the largest vertex degree of $G(\texttt{cand})$ is large enough. We define the largeness of a vertex degree in a graph by $p = \frac{d}{n}$, in which n represents the total number of vertices in the graph and d represents a vertex degree. Specifically, if the largest degree of the vertices in $G(\texttt{cand})$, compared with the total number vertices in the graph, exceeds a predefined threshold θ (e.g., $\theta = 0.8$), the hybrid algorithm would execute the multi-way partitioning operation; otherwise, it would execute the binary partitioning operation. As shown in Sect. 5.3, the value of the threshold θ has only marginal influence on the performance of the hybrid algorithm if it is set between 0.6 and 0.8. We suggest that it is set to be 0.8 in practical implementation.

The hybrid algorithm is sketched in Algorithm 3. Lines 5–19 specify the multi-way partitioning operation and Lines 22–28 specify the binary partitioning operation. The operation of multi-way partitioning is similar to the core search operation used by the *BK* algorithm. The difference is that it uses the `not` set to filter out unnecessary search subtrees (Line 12). Once $G(\texttt{cand})$ meets the condition specified at Line 5, it is partitioned into multiple subgraphs, each of which invokes a new recursive function (Line 13). After that, the current function terminates its execution (Line 20). Otherwise, $G(\texttt{cand})$ is partitioned into $G(\texttt{cand}^+)$ and $G(\texttt{cand}^-)$. A new recursive function is invoked to process $G(\texttt{cand}^+)$ (Line 27). The subgraph $G(\texttt{cand}^-)$ is instead iteratively partitioned until it becomes clique (Line 4 and Lines 29–30).

Based on the correctness proofs of the *BK* and *GP* algorithms, it can be easily shown that Algorithm 3 exactly returns all the maximal cliques in G. On the space and time

complexity of Algorithm 3, we have Theorem 4, whose proof is presented in "Appendix 3".

Theorem 4 *Given a connected graph $G = (V, E)$, Algorithm 3 has the space complexity of $O(|E|)$ and the time complexity of $O(|E|\mu(G))$, in which $\mu(G)$ represents the number of different cliques in G.*

3.4 Notes on Implementation

The program reads a graph G into memory, and then iteratively computes the maximal cliques of every vertex in G. We store the vertices in the original graph G in an array and their adjacency lists as hash sets. Similarly, all the `cand` sets are maintained by hash sets. As a result, the intersection of two vertex sets can be performed by hash look-ups. Clique verification is achieved by checking vertex degrees.

For the Hybrid algorithm, vertex degrees in each subgraph resulting from a multi-way BK partitioning operation are computed by intersecting two adjacency sets. For the GP algorithm, the degree of a vertex v_i in $\texttt{cand}^+$ of G_v^+ is computed by intersecting the adjacency set of v_i with the $\texttt{cand}^+$ set. For the vertices in the $\texttt{cand}^-$ set of G_v^-, only those connected to v needs to decrease their degrees by 1. Selecting a partitioning anchor with the minimal degree in `cand` however requires $O(|\texttt{cand}|)$ time because it has to sequentially scan all the vertices in the hash set. To enable more efficient anchor selection, we also maintain a degree map, in which the vertex degrees of `cand` are stored as a sorted linked list and each entry in the degree list has a corresponding vertex list consisting of all the vertices with the specified degree. The degree map of the G_v^- subgraph is inherited from that of its parent with corresponding updates while the degree map of G_v^+ is constructed from scratch. With the degree map, selecting a partitioning anchor in `cand` only involves picking up a vertex in the vertex list of the first entry in the degree list. It takes only constant time.

4 Parallel Solutions

4.1 General Procedure

The parallel solution consists of two steps. In the first step, for every vertex v in the graph G, it retrieves an induced subgraph of G whose vertices are relevant to the computation of v's maximal cliques. In the second step, it performs iterative graph partitioning on every vertex. Both subgraph retrieval and clique computation on individual vertices are distributed across multiple computing nodes.

We observe that the computational workload on the vertices may be unbalanced: the computation on a vertex may be more expensive than on another because it has a larger search space. In case that the computation on a vertex is too time-consuming, it becomes a parallel performance bottleneck. A good property of the GP approach is that it enables easy and effective load balancing. Since GP iteratively partitions a large G_v into a series of small graphs, whose computations are independent, the computation on a vertex can be easily parallelized. In practice, recursive function usually takes only a few iterations (no more than 3–4 iterations in our experiments in Sect. 5.2) to transform a big G_v into many sufficiently small subgraphs. With sufficiently small tasks, effective load balancing can be achieved by sending some tasks on a computing node with heavy workload to another with lighter workload.

To achieve workload balance, the procedure iteratively invokes the Compute–Shuffle cycle. In the Compute phase, every computing node performs the partitioning operation on the subgraphs it has received; in the Shuffle phase, all the intermediate subgraphs on the nodes are reshuffled so that every node receives roughly the same number of them. The workload limit of each Compute phase can be quantified by the consumed CPU time.

In general, the parallel procedure consists of the following two steps:

1. *Subgraph retrieval* For every vertex v in the graph G, retrieve the induced graph G_v consisting of v and its neighboring vertices in G;
2. *Iterative computation*
   - *Compute phase* For each computing node, sequentially compute the maximal cliques of its assigned subgraphs by the GP or hybrid algorithm;
   - *Shuffle phase* Evenly reshuffle all the intermediate subgraphs across the nodes;

4.2 MapReduce Solutions

This subsection describes the MapReduce solutions based on the GP and hybrid sequential algorithms. Based on the observation that non-trivial cliques consist of triangles, we uses the technique of triangle enumeration proposed in [12], which is more efficient than 2-hop retrieval, to implement the process of subgraph retrieval.

Algorithm 4: The Computation at Reducer based on the GP Algorithm

```
   Input: A queue of unfinished subgraphs Q;
1  while (Q is not empty) and (workload limit has not been reached) do
2      Dequeue a subgraph G_u from Q;
3      while (G_u is not a clique) do
4          Choose the vertex w with the minimal degree in G_u as the anchor;
5          Partition G_u into G_w^+ and G_w^-;
6          if |cand(G_w^+)| ≤ k then
7              if (workload limit has been reached) then
8                  Enqueue(G_w^+, Q);
9              else
10                 Process G_w^+ using Algorithm 2 to the end;
11         else
12             Enqueue(G_w^+, Q);
13         G_u=G_w^-;
14     if (G_u can not be pruned) then
15         Output G_u;
```

Algorithm 5: The Function of $Enqueue(G, Q)$

```
1  if G can not be pruned then
2      if G is a clique then
3          Output G;
4      else
5          Enqueue G into Q;
```

The program of iterative computation consists of a series of MapReduce cycles. In the Map phase, the mappers reads the unfinished subgraphs and randomly map them to reducers such that each reducer receives roughly the same number of subgraphs. In the Reduce phase, the reducers enumerate the maximal cliques of their assigned subgraphs by sequential algorithms. The MapReduce cycle is iteratively invoked until no unfinished subgraph is left.

The computation at a reducer based on the GP algorithm is sketched in Algorithm 4. Maintaining the subgraphs by a queue Q, it iteratively dequeues a subgraph G_u from the queue for graph partitioning. If the resulting G_w^+ has a small size, which means that its maximal clique computation can be finished in short time, it is iteratively partitioned to the end (Lines 7–10). Otherwise, it is temporarily enqueued into Q if it is not a clique (Line 12). It then iteratively partitions G_w^- in the same manner as G_u (Line 13). The operations of subgraph dequeue and graph partitioning are iteratively performed until the queue becomes empty or a predefined workload limit is reached.

Algorithm 6: The Computation at Reducer based on the Hybrid Algorithm

```
   Input: A queue of unfinished subgraphs Q;
1  while (Q is not empty) and (workload limit has not been reached) do
2      Dequeue a subgraph G_u from Q;
3      while (G_u is not a clique) do
4          if (the largest degree of the vertices in G_u exceeds θ) then
5              w ← the vertex with the largest degree in G_u;
6              Partition G_u into G_v and {G_w1,...,G_wh};
7              for (1 ≤ i ≤ h) do
8                  if |G_wi| ≤ k then
9                      if (workload limit has been reached) then
10                         Enqueue(G_wi, Q);
11                     else
12                         Process G_wi using Algorithm 3 to the end;
13                 else
14                     Enqueue(G_wi, Q);
15             if (G_v can not be pruned) then
16                 G_u=G_v;
17         else
18             Choose the vertex v with the minimal degree in G_u as the anchor;
19             Partition G_u into G_v^+ and G_v^-;
20             if |cand(G_v^+)| ≤ k then
21                 if (workload limit has been reached) then
22                     Enqueue(G_v^+, Q);
23                 else
24                     Process G_v^+ using Algorithm 3 to the end;
25             else
26                 Enqueue(G_v^+, Q);
27             G_u=G_v^-;
28     if (G_u can not be pruned) then
29         Output G_u;
```

The computation at a reducer based on the Hybrid algorithm, as sketched in Algorithm 6, is similar. If the largest vertex degree of a dequeued subgraph G_u exceeds the threshold of θ, it executes the operation of multi-way partitioning (Lines 5–16); otherwise, it executes the operation of binary partitioning (Lines 18–27). To ensure that G_u can be divided into many small subgraphs in a single reduce phase, the algorithm iteratively processes G_v resulting from the multi-way partitioning operation until it becomes a clique (Line 3) or it can be pruned (Lines 15–16).

5 Experimental Evaluation

This section empirically evaluates the performance of our proposed approaches by a comparative study. Since the BK algorithm has been widely reported to be faster than its alternatives, we compare our approach with the state-of-the-art implementation of the BK algorithm [28]. The typical parallel approach based on BK confines the computation on a vertex to a computing node. We enhance the parallel BK approach with the dynamic load balancing proposed in [32]. It was originally implemented by MPI in [32]. We have instead implemented a MapReduce version. Each reducer is set to have a predefined workload limit. After every reducer reaches its workload limit, the unfinished subgraphs are evenly redistributed across computing nodes. All our implementations have been made open source. They can be downloaded at [16].

Our experiments are conducted on both real and synthetic graph datasets. The evaluation on real datasets can show the efficiency of the proposed algorithms in real applications, while the evaluation on synthetic datasets can easily demonstrate their sensitivity to varying graph characteristics. Two synthetic datasets are generated by the SSCA#2 generator [2] and the power-law generator R-MAT [3] respectively. A SSCA#2 graph is directed, and made up of random-sized cliques, with a hierarchical inter-clique distribution of edges based on a distance metric. We vary the values of the *TotVertices* and *MaxCliqueSize* parameters, which specify the number of vertices and the size of the maximum clique respectively. The R-MAT generator applies the Recursive Matrix (R-MAT) graph model to produce the graphs with power-law degree distributions and small-world characteristics, which are common in many real life graphs. We vary two parameter values, the number of vertices and the number of edges. The real graphs, which are selected from [29], are in various domains including communication networks, social networks, web graphs and protein networks. The details of test datasets are summarized in Table 1.

For the hybrid algorithm, we set the largest degree threshold θ in Algorithm 3 to 0.8 in the comparative study of Sects. 5.1 and 5.2. Our evaluation in Sect. 5.3 shows that if set between 0.0 and 0.8, the value of the threshold θ has only marginal influence on the performance of the hybrid algorithm.

Sequential algorithms are evaluated on a desktop with memory size of 16G and 6 Intel Core i7 CPU with the frequency of 3.3GHz. Parallel evaluation are conducted on a 13-machine cluster. Each machine runs the Ubuntu Linux (version 10.04) and has memory size of 16G, disk storage of 160G and 16 Intel Xeon E5502 CPUs with the frequency of 1.87GHz. The parallel solutions based on MapReduce are implemented on Hadoop (version 0.20.2) [17]. Each experiment is run three times and its running time averaged. We observe that time difference between different runnings does not exceed 10% of the total consumed time.

5.1 Evaluation of Sequential Algorithms

In this subsection, we evaluate the performance of the sequential algorithms on both real and synthetic graphs. Performance is evaluated on the metric of runtime.

5.1.1 On Real Datasets

The evaluation results on the real graphs are presented in Table 2. Note that running the Twitter dataset is beyond

Table 1 Details of the real and synthetic graph datasets

Dataset	Data description	Number of vertexes	Number of edges
EuAll	Email network from a EU Research Institution	265,214	364,481
WebGoogle	Web graph from Google	875,713	4,322,051
BerkStan	Web graph of Berkeley and Stanford	685,230	7,600,595
WikiComm	Wikipedia communication network	1,928,669	3,494,674
Pokec	Pokec online social network	1,632,803	30,622,564
Protein-1	A protein network	5816	313,628
Protein-2	A protein network	8176	457,991
WikiTalk	A social network	2,394,385	4,659,565
Skitter	Autonomous systems graphs	1,696,415	11,095,298
Twitter	Social circles from Twitter	11,316,811	85,331,846
R-MAT	Synthetic graphs with power-law degree distributions and small-world characteristics	Two parameters used: the number of vertices and the ratio of edges to vertices	
SSCA#2	Synthetic graphs with a hierarchical inter-clique distribution of edges based on a distance metric	Two parameters used: the number of vertices and the size of maximum clique	

Table 2 Evaluation of sequential algorithms on real graphs

Runtime(s)	EuAll	WebGoogle	BerkStan	WikiComm	Pokec	Protein-1	Protein-2	WikiTalk	Skitter
BK	1.56	8.94	28.79	388.70	**55.73**	115.76	216.53	9247.08	1720.48
GP	1.00	**8.78**	**17.57**	42.41	105.86	221.16	660.73	898.01	686.79
Hybrid	**0.97**	9.04	32.35	**35.19**	103.05	**103.06**	**154.27**	**554.64**	**359.26**

The bold values represent the minimal runtime consumed by the three approaches

the capability of a single machine. Therefore, they will be used later for parallel evaluation.

It can be observed that GP achieves overall better performance than BK. On some datasets (e.g., WikiComm and WikiTalk), GP runs roughly 10 times faster than BK. On the datasets where GP performs worse than BK (e.g., Berkstan and Protein-1), their performance difference is much smaller. It can also be clearly observed that the hybrid algorithm achieves the best performance among them. On most test datasets, Hybrid consumes the least runtime. It is worthy to point out that the outperformance margins achieved by Hybrid are considerable on many test datasets. For instance, on the Skitter dataset, GP takes around 40% of the runtime consumed by BK and Hybrid further cuts runtime by around 50%.

Our experiment show that the BK algorithm is very sensitive to particular graph characteristics. Its performance is usually very volatile. In comparison, GP's performance is more stable. The Hybrid algorithm achieves the best and most stable performance by effectively leveraging the advantages of the BK and GP algorithms.

5.1.2 On Synthetic Datasets

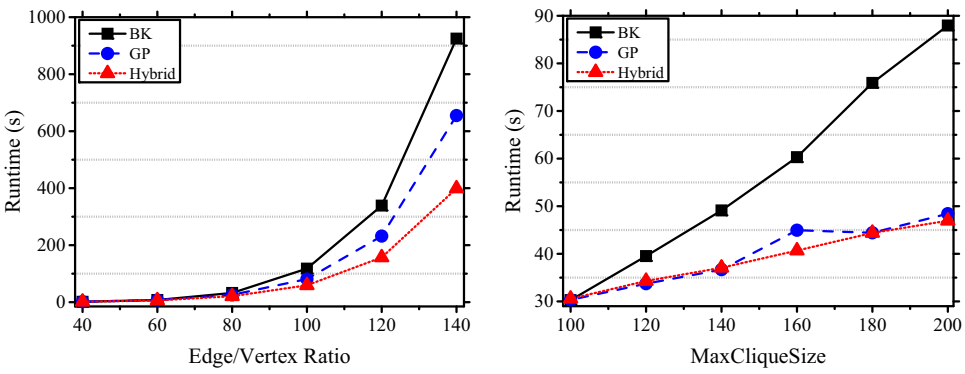
On synthetic datasets, we aim to investigate the comparative performance of different algorithms and how their performance varies with graph characteristics. On R-MAT graphs, the number of vertices is set to be 5000 and the edge-to-vertex ratio varies from 40 to 140. On SSCA graphs, the number of vertices is set to be 2^{20} and the size of the maximum clique varies from 100 to 200.

The evaluation results are presented in Fig. 2. On R-MAT, GP performs better than BK and Hybrid performs better than GP. On SSCA, Hybrid and GP achieve similar performance and both of them perform better than BK. It is interesting to note that the outperformance margins of GP and Hybrid over BK steadily increase with graph density. Similar to what were observed in the evaluation on real graphs, our results on synthetic datasets demonstrate that compared with GP and Hybrid, BK is much more sensitive to particular graph characteristics (e.g., graph density and sizes of maximal cliques).

5.2 Evaluation of Parallel Solutions

In this subsection, we evaluate the performance of different approaches, comparing GP and Hybrid against BK, on the Twitter dataset. Since all the parallel solutions use the same method of subgraph retrieval, we exclude its cost from performance evaluation in our study. We specify the parameter k in Algorithms 4 and 6 by the number of vertices contained by a graph. It is set to be 80. The maximal execution time per reduce phase is set to be 300 s. The workload limit of reduce phase is similarly set for the BK approach.

On the synthetic RMAT and SSCA graphs, the parallel performance of different approaches is similar to what are observed in sequential evaluation. Their detailed evaluation results are thus omitted here. We present the evaluation results on the largest real graph, Twitter. Note that processing the entire Twitter graph takes too long even on our machine cluster. We therefore generate 5 random test tasks,



(a) R-MAT Graphs **(b)** SSCA Graphs

Fig. 2 Evaluation of sequential algorithms on R-MAT and SSCA datasets

Table 3 Parallel evaluation based on Hadoop on Twitter

	MapReduce cycles			Runtime (s)		
	BK	GP	Hybrid	BK	GP	Hybrid
D_T^1	3	2	2	775	441	**423**
D_T^2	3	3	3	944	917	**728**
D_T^3	4	3	2	1278	707	**486**
D_T^4	16	3	3	5455	**711**	772
D_T^5	18	4	3	5760	1111	**1054**

The bold values represent the minimal runtime consumed by the three approaches

denoted by $D_T^1, \ldots, D_T^5$, by choosing some vertices with large degrees in the graph for evaluation purpose. The maximal cliques of the chosen vertices are computed over the entire graph. The maximal vertex degree in the Twitter graph is more than one million. We randomly choose 5 vertices with degrees of more than 500,000 for each test task.

The comparative results on Twitter are presented in Table 3. Similar to what were observed in sequential evaluation, the performance of BK is very volatile. On some test tasks (e.g., D_T^2), the performance of BK is similar to that of GP. On other test tasks (e.g., D_T^4 and D_T^5), it performs significantly worse than GP. GP achieves overall better performance than BK. It can also be observed that Hybrid performs better than both BK and GP. On some test tasks (e.g., D_T^2 and D_T^3), its outperformance margins over GP are considerable.

5.3 Hybrid: Varying the Threshold θ

Our experiment evaluates the performance variation of the hybrid algorithm with the value of θ. We set the value of θ to be 0.6, 0.7 and 0.8. The detailed results on some of the real graphs listed in Table 1 are presented in Table 4. The results on other datasets are similar, thus omitted here. It can be observed that in the range of [0.6,0.8], the value of θ can only marginally influence the performance of the hybrid algorithm. This observation bodes well for the efficacy of the hybrid algorithm in real applications.

6 Related Work

Maximal clique enumeration have been studied extensively in the literature [4, 8–10, 25, 33, 34]. Due to its NP-Completeness, existing work focused on efficient search. Most of the proposed approaches were based on the classical BK algorithm [4], which has been widely reported as being faster than its alternatives [5, 15]. Authors of [10] proposed an efficient algorithm, which was also based on BK search, for maximal clique enumeration with limited memory. Authors of [8, 9] proposed to speed up clique detection by indexing the core structures of a special type of graph called H*-graph. Instead of BK, another approach [11, 24, 36] uses the strategy of *reverse search*. The key feature of this approach is that it is possible to define an upper bound on their runtime as a polynomial with respect to the number of maximal cliques in a graph. Note that focusing on centralized search, the efficient implementations of existing algorithms usually rely on global state and cannot be easily parallelized. There are also some work [30, 31] studying the closely related problem of detecting maximum clique. The algorithms they used are, however, variants of the BK algorithm.

Table 4 Influence of the parameter θ on hybrid

Runtime(s)	EuAll	WebGoogle	Berkstan	WikiComm	Pokec
$p = 0.8$	0.97	9.04	32.35	35.19	103.05
$p = 0.7$	0.97	9.14	32.95	34.48	105.48
$p = 0.6$	0.99	9.24	32.65	35.98	103.85

Due to the increasing popularity of the MapReduce framework, the solutions have been proposed to parallelize maximal clique detection on MapReduce [13, 23, 38]. They proposed to distribute the vertices across workers and compute every vertex's maximal cliques in parallel. On the core algorithm for efficient search, they, however, used the BK algorithm or its variants. Authors of [39] proposed a fault-tolerant parallel solution for maximum clique detection based on MapReduce. It also used the BK algorithm for efficient search. A parallel solution for maximal clique enumeration based on MPI has been proposed in [32]. It proposed a dynamic load balancing technique that enabled an idle worker to "steal" workload from another busy worker. As we showed in Sect. 5.2, limited by the efficiency of BK search, its performance was still quite sensitive to graph characteristics.

Orthogonal to our work, many works extended the definition of clique to other dense subgraph structures (e.g., maximal cliques in an uncertain graph [42], cross-graph quasi-cliques [21], k-truss [20], and densest-subgraph [35]), and studied their applications. The existing algorithms for these problem are centralized. The search process of these dense structures is usually NP-Complete, thus computationally expensive over massive real graphs. However, efficient parallelization of their search processes over a machine cluster remains an open question.

7 Conclusion

In this paper, we propose a novel approach based on binary graph partitioning for maximal clique enumeration over graph data. Compared with the state-of-the-art BK

approach, it can effectively divide a graph into many small tasks with less iterations. We also present a hybrid approach that can effectively leverage the advantages of both BK and GP approaches. We develop efficient sequential algorithms as well as corresponding parallel solutions. Finally, our extensive experiments on real and synthetic graph data demonstrate the performance advantage of our proposed solutions over the state-of-the-art ones.

Appendix 1: Proof of Theorem 2

Proof Firstly, if without the pruning operations specified in Lines 9 and 13 of Algorithm 2, all the maximal cliques in G are contained in the set of cliques returned by Algorithm 2.

Secondly, if there exists a vertex in the current $\texttt{not}$ set that is connected to all the vertices in the current $\texttt{cand}$ set, the recursive function cannot generate any new maximal clique. Consider a clique C_i in the graph $G(\texttt{anchor} \cup \texttt{cand})$. Suppose that the vertex u in the $\texttt{not}$ set is connected to all the vertices in the $\texttt{cand}$ set. Note that Algorithm 2 ensures that every vertex in $\texttt{not}$ is connected to all the vertices in the $\texttt{anchor}$ set. As a result, u is connected to all the vertices of C_i. Therefore, the clique C is not maximal.

Finally, any clique returned by Algorithm 2 is maximal. Assume that it returns two cliques, C_1 and C_2, and C_1 is contained by C_2. Suppose that C_1 consists of k vertices, $\{v_1, v_2, \ldots, v_k\}$, and C_2 has an additional vertex u. Also suppose that C_1 is generated by combining the $\texttt{anchor}_1$ set and the $\texttt{cand}_1$ set. Since the vertex u is not in $\texttt{anchor}_1$ but connected to all the vertices in $\texttt{anchor}_1$, its exclusion from $\texttt{cand}_1$ should be a result of a previous graph partitioning operation with u as the partitioning anchor. Therefore, the vertex u should be included in the $\texttt{not}$ set of the corresponding partitioned graph $\mathrm{G}(\texttt{anchor}^- \cup \texttt{cand}^-)$, whose recursive partitioning later generates the clique C_1. Since u is connected to all the vertices in $\texttt{anchor}_1$, Algorithm 2 ensures that it is in the $\texttt{not}$ set of the partitioned graph $\mathrm{G}(\texttt{anchor}_1 \cup \texttt{cand}_1)$. With u being connected to all the vertices in $\texttt{cand}_1$, Algorithm 2 should have filtered C_1 out. Contradiction. □

Appendix 2: Proof of Theorem 3

Proof We first analyze its space complexity. It iteratively partitions the $G(\texttt{cand}^-)$ branch until $G(\texttt{cand}^-)$ becomes a clique. Besides the $G(\texttt{cand}^-)$ graph, it also has to store the resulting $G(\texttt{cand}^+)$ subgraphs in a stack S. Each $G(\texttt{cand}^+)$ results from a partitioning operation with a vertex v_i as anchor. Note that the first-in-last-out operation order of stack ensures that each $G(\texttt{cand}^+)$ subgraph in the stack S has a distinct partitioning anchor. Since each vertex in the $\texttt{anchor}^+$, $\texttt{cand}^+$ and $\texttt{not}^+$ sets of $G(\texttt{cand}^+)$ (except the vertex v_i itself) should be connected to v_i, the required space to store $G(\texttt{cand}^+)$ is bound by $O(|E_i|)$, in which E_i represents the set of edges with v_i as one of its end points. As a result, the required space to store all $G(\texttt{cand}^+)$ branches is bound by $O(|E|)$. It follows that the space complexity of Algorithm 2 is $O(|E|)$.

Secondly, we analyze its time complexity. Consider a variant of Algorithm 2 without the pruning operation specified on Line 9. Obviously, its time complexity is an upper bound on the time complexity of Algorithm 2. The traversal tree generated by the recursive function without pruning is a binary tree, in which each internal node has exactly two children. Since the cliques generated by the $G(\texttt{cand}^+)$ branch are guaranteed to be different from those generated by the $G(\texttt{cand}^-)$ branch, each leaf node corresponds to a different clique (maximal or non-maximal). Therefore, the size of the binary tree is bounded by $O(\mu(G))$. Accordingly, the total number of invoked graph partitioning operations is bounded by $O(\mu(G))$. Since each invocation of graph partitioning requires $O(|E|)$ time, the time complexity of the recursive function is $O(|E|\mu(G))$. □

Appendix 3: Proof of Theorem 4

Proof The space complexity analysis of Algorithm 3 is similar to that of Algorithm 2. Each subgraph recorded for later processing can be considered to correspond to a different anchor. Therefore, its required space is bounded by $O(|E|)$.

Secondly, we analyze its time complexity. Consider a variant of Algorithm 3 without the pruning operation by the $\texttt{not}$ set. Obviously, its time complexity is an upper bound on the time complexity of Algorithm 3. It would generate different cliques (maximal or non-maximal). Also note that in its traversal tree, each leaf corresponds to a different clique and each internal node has at least two children. The size of its traversal tree is thus bounded by $O(\mu(G))$. Therefore, the time complexity of Algorithm 3 is bounded by $O(|E|\mu(G))$. □

References

1. Akkoyunlu EA (1973) The enumeration of maximal cliques of large graphs. SIAM J Comput 2(1):1–6
2. Bader DA, Madduri K (2005) Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In: In Proceedings of 12th international conference on high performance computing, pp 465–476
3. Bader DA, Madduri K (2006) Gtgraph: a synthetic graph generator suite, pp 1–4. http://www.cse.psu.edu/~madduri/software/GTgraph/
4. Bron C, Kerbosch J (1973) Algorithm 457: finding all cliques of an undirected graph. Commun ACM 16(9):575–577
5. Cazals F, Karande C (2008) A note on the problem of reporting maximal cliques. Theor Comput Sci 407(1):564–568
6. Cazals F, Karande C (2008) A note on the problem of reporting maximal cliques. Theor Comput Sci 407(1–3):564–568
7. Chen Q, Fang C, Wang Z, Suo B, Li Z, lves ZG (2016) Parallelizing maximal clique enumeration over graph data. In: DASFAA, pp 249–264
8. Cheng J, Ke Y, Fu AW-C, Yu JX, Zhu L (2011) Finding maximal cliques in massive networks. ACM Trans. Datab. Syst. 36(4):1–34
9. Cheng J, Ke Y, Fu AW, Zhu L (2010) Finding maximal cliques in massive networks by h*-graph. In: SIGMOD, pp 447–458
10. Cheng J, Zhu L, Chu YKS (2012) Fast algorithms for maximal clique enumeration with limited memory. In: KDD, pp 1240–1248
11. Chiba N, Nishizeki T (1985) Arboricity and subgraph listing algorithms. SIAM J Comput 14(1):210–223
12. Cohen J (2009) Graph twiddling in a mapreduce world. Comput Sci Eng 11(4):29–41
13. Du N, Wu B, Xu LT, Wang B, Pei X (2006) A parallel algorithm for enumerating all maximal cliques in complex network. In: ICDM workshops, pp 320–324
14. Eppstein D, Löffler M, Strash D (2010) Listing all maximal cliques in sparse graphs in near-optimal time. In: ISAAC(1), pp 403–414
15. Eppstein D, Strash D (2011) Listing all maximal cliques in large sparse real-world graphs. In: 10th International symposium on experimental algorithms, pp 364–375
16. GP Project: efficient maximal clique and k-plex detection over graph data. http://www.wowbigdata.cn/gp/clique.html
17. Hadoop: an open-source implementation of mapreduce. http://hadoop.apache.org/
18. Hanneman R (2005) Introduction to social network methods, chapter 11: cliques. http://faculty.ucr.edu/~hanneman/nettext/
19. Haraguchi M, Okubo Y (2006) A method for pinpoint clustering of web pages with pseudo-clique search. In: Jantke K, Lunzer A, Spyratos N, Tanaka Y (eds) Federation over the web, volume 3847 of lecture notes in computer science. Springer, Berlin, pp 59–78
20. Huang X, Cheng H, Qin L, Tian W, Yu JX (2014) Querying k-truss community in large and dynamic graphs. In: SIGMOD, pp 1311–1322
21. Jiang DX, Pei J (2009) Mining frequent cross-graph quasi-cliques. TKDE 2(4):1–42
22. Leskovec J, Lang KJ, Dasgupta A, Mahoney MW (2008) Statistical properties of community structure in large social and information networks. In: WWW, pp 695–704
23. Lu L, Gu Y, Grossman R (2010) dmaximalcliques: A distributed algorithm for enumerating all maximal cliques and maximal clique distribution. In: IEEE international conference on data mining workshops, pp 1320–1327
24. Makino K, Uno T (2004) New algorithms for enumerating all maximal cliques. In: SWAT, lecture notes in computer science, vol 3111, pp 260–272
25. Modani N, Dey K (2008) Large maximal cliques enumeration in sparse graphs. In: CIKM, pp 1377–1378
26. On B-W, Elmacioglu E, Lee D, Kang J, Pei J (2006) Improving grouped-entity resolution using quasi-cliques. In: ICDM, pp 1008–1015
27. Pavlopoulos GA, Secrier M, Moschopoulos CN, Soldatos TG, Kossida S, Aertes J, Schneider R, Bagos PG (2011) Using graph theory to analyze biological networks. BioData Min 4:10
28. Quick Cliques: quickly compute all maximal cliques in sparse graphs. https://github.com/darrenstrash/quick-cliques
29. Real graph datasets. http://snap.stanford.edu/data/
30. Rossi RA, Gleich DF, Gebremedhin AH, Patwary MMA (2014) Fast maximum clique algorithms for large graphs. In: WWW, pp 365–366
31. Rossi RA, Gleich DF, Gebremedhin AH (2015) Parallel maximum clique algorithms with applications to network analysis. SIAM J Sci Comput 37(5):589–616
32. Schmidt MC, Samatova NF, Thomas K, Park BH (2009) A scalable, parallel algorithm for maximal clique enumeration. J Parallel Distrib Comput 69(4):417–428
33. Stix V (2004) Finding all maximal cliques in dynamic graphs. Comput Optim Appl 2:173–186
34. Tomita E, Tanaka A, Takahashi H (2006) The worst-case time complexity for generating all maximal cliques and computational experiments. Theor Comput Sci 363(1):28–42
35. Tsourakakis C, Bonchi F, Gionis A, Gullo F, Tsiarli M (2013) Denser than the densest subgraph: Extracting optimal quasi-cliques with quality guarantees. In: KDD, pp 104–112
36. Tsukiyama S, Ide M, Shirakawa I (1977) A new algorithm for generating all the maximal independent sets. SIAM J Comput 6(3):505–517
37. Wang J, Zeng Z, Zhou L (2006) Clan: An algorithm for mining closed cliques from large dense graph databases. In: ICDE, pp 73–82
38. Wu B, Yang S, zhao H, Wang B (2009) A distributed algorithm to enumerate all maximal cliques in mapreduce. In: International conference on frontier of computer science and technology, pp 45–51
39. Xiang JG, Guo C, Aboulnaga A (2013) Scalable maximum clique computation using mapreduce. In: ICDE, pp 74–85
40. Yang S, Wang B, zhao H, Wu B (2009) Efficient dense structure mining using mapreduce. In: IEEE international conference on data mining workshops, pp 332–337
41. Zhang Y, Abu-Khzam FN, Baldwin NE, Chesler EJ, Langston MA, Samatova NF (2005) Genome-scale computational approaches to memory-intensive applications in systems biology. In: ACM/IEEE supercomputing, pp 12–12
42. Zou ZN, Li JZ, Gao H, Zhang S (2010) Finding top-k maximal cliques in an uncertain graph. In: ICDE, pp 649–652